Core Java: Collections And Generics

Persistent Interactive | Persistent University

Objectives :

- Choosing appropriate collections
- Queue and Dequeue interface
- NavigableSet and NavigableMap
- Enumerations, Iterators and ListIterators
- Equals and HashCode
- Comparable and Comparator
- Generics – Bounded Types
- Generics with Wildcards
- Type Inference (<> operator)

Queue Interface

- Represents ordered list of objects just like list but intended to use slightly different.
- Elements are inserted at the end and removed from the beginning.
- Queue works on FIFO(First in first out) basis.
- We can choose from following Queue implementations:-
  - LinkedList   - Standard Queue Implementation
  - PriorityQueue – Stores elements according to natural order (If implements comparable), or comparator passed to the PriorityQueue.

Methods of Queue…

Method	Description
add(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
element()	Retrieves, but does not remove, the head of this queue.
offer(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty.
remove()	Retrieves and removes the head of this queue.

Persistent

PriorityQueue……quick preview

- Construct an empty priority queue
- Construct a priority queue with the specified capacity
- Construct a priority queue with the specified capacity and a custom comparator used for ordering elements. By default elements are ordered by their natural ordering.

- Add values to the queue through *add*( ) method. Throws exception on failure.
- Add values through the *offer( )* method. Returns false on failure.
- Remove value at the head.

```
PriorityQueue<Long> priorityQueue = new
PriorityQueue<Long>();

// PriorityQueue<Long> priorityQueue = new
PriorityQueue<Long>(10);

// PriorityQueue<Long> priorityQueue = new
PriorityQueue<Long>(10, new QueueOrder());

priorityQueue.add(500L);
priorityQueue.add(200L);
priorityQueue.add(400L);
priorityQueue.add(100L);

System.out.println(priorityQueue.offer(700L));

System.out.println(priorityQueue.poll());
```

Deque interface

- A Double Ended Queue.
- Extends java.util.Queue interface.
- Allows elements to be added & removed from both the ends.
- ArrayDeque & LinkedList classes implement Deque interface.

Methods in Deque interface

- Methods that throw exception when operation fails.
- Methods used for insert, remove & examine first element.

Method	Description
addFirst(e)	Inserts the specified element at the front of the deque.
removeFirst()	Retrieves and removes the first element of the deque.
getFirst()	Retrieves, but does not remove, the first element of the deque.
addLast(e)	Inserts the specified element at the end of deque.
removeLast()	Retrieves and removes the last element of deque.
getLast()	Retrieves, but does not remove, the last element of deque.

Methods in Deque interface

- Methods return false or null value when operation fails.
- Methods used for insert, remove & examine first element.

Method	Description
offerFirst(e)	Inserts the specified element at the front of deque unless it would violate capacity restrictions.
pollFirst()	Retrieves and removes the first element of deque, or returns null if deque is empty.
peekFirst()	Retrieves, but does not remove, the first element of deque, or returns null if deque is empty.
offerLast(e)	Inserts the specified element at the end of deque unless it would violate capacity restrictions.
pollLast()	Retrieves and removes the last element of deque, or returns null if deque is empty.
peekLast()	Retrieves, but does not remove, the last element of deque, or returns null if deque is empty.

ArrayDeque class

- Implements Deque interface
- Stores the elements in an array internally.
- Grows as needed even though elements are stored in array.
- No capacity restriction.
- Null values can not be added.
- Not thread safe.

NavigableSet interface

- Extends SortedSet interface.
- Used to handle retrieval of elements based on closest-match searches.
- Have navigation methods along with sorting functionality.
- TreeSet class implements this interface.

Methods in NavigableSet interface

Method name	Description
ceiling(E e)	Returns the least element in this set greater than or equal to the given element else null value if such no element found.
descendingIterator()	Returns an iterator over the elements in the set, in descending order.
descendingSet()	Returns a reverse order view of the elements contained in this set.
floor(E e)	Returns the greatest element in this set less than or equal to the given element else null value if no such element found.
headSet(E toElement)	Returns a view of the portion of this set whose elements are strictly less than toElement.
higher(E e)	Returns the least element in this set strictly greater than the given element, or null if there is no such element.

Methods in NavigableSet interface

Method name	Description
lower(E e)	Returns the greatest element in this set less than the given element, or null if there is no such element.
pollFirst()	Retrieves and removes the lowest element and returns null if the set is empty.
pollLast()	Retrieves and removes the highest element and returns null if the set is empty.
subSet(E fromElement, E toElement)	Returns a view of the portion of the set whose elements range from fromElement, inclusive, to toElement, exclusive.
tailSet(E fromElement)	Returns a view of the portion of the set whose elements are greater than or equal to fromElement.

NavigableMap

- Supports retrieval of entries based on closest match to a given key or keys.
- Extends SortedMap interface
- TreeMap class implements this interface.
- Have navigation methods along with sorting functionality.
- Elements can be accessed & traversed in either ascending or descending order.

Methods in NavigableMap interface

Method name	Description
ceilingKey(K key)	Returns the least key greater than or equal to the given key, or null if there is no such key.
descendingKeySet()	Returns a reverse order NavigableSet view of the keys contained in the map.
descendingMap()	Returns a reverse order view of the mappings contained in the map.
headMap(K toKey)	Returns a view of the portion of the map whose keys are strictly less than toKey.
tailMap(K fromKey)	Returns a view of the portion of the map whose keys are greater than or equal to fromKey.
subMap(K fromKey, K toKey)	Returns a view of the portion of the map whose keys range from fromKey, inclusive, to toKey, exclusive.

Persistent

Iteration order for the implementations

- The criteria to choose the collection

Implementation	Iteration order
ArrayList	Insertion Order
LinkedList	Insertion Order
HashSet	Random order
LinkedHashSet	Insertion Order
TreeSet	Ascending order based on Comparable / Comparator
HashMap	Random Order
LinkedHashMap	Insertion Order as per Key
TreeMap	Ascending order of keys based on Comparable / Comparator

Enumerations and Iterators

- Both interfaces are used to iterate over indexed as well as non indexed implementations.
- Got methods to perform these operations.
- Iterator has facility to remove the element at the time of iterating over a collection implementation.

Enumeration Interface	Iterator Interface
hasMoreElement()	hasNext()
nextElement()	next()
N/A	remove()

The collections framework……...iterators

Iterator

An forward only iterator over a collection

ListIterator

A bi-directional iterator over a collection

An ArrayList is Iterable with the “foreach“ loop

An ArrayList is accessible with methods from the Iterator interface

```
ArrayList<String> sObj = new ArrayList<String>();

sObj.add("Great");
sObj.add("Cool");
sObj.add("Fine");
sObj.add("Nice");

for(String s : sObj) System.out.println(s);
Iterator<String> itr = sObj.iterator();

while(itr.hasNext())
System.out.println(itr.next());
```

Use of equals and hashcode methods

- equals() and hashcode() methods are defined in Object class.
- The equals() method is used to check the equality of two objects.
- The hashcode() method of Object class returns integer value. That determines the integer representation of memory address where the object is stored.
- Whenever **a.equals(b)**, then *a.hashCode()* must be same as *b.hashCode()*.
- If any one of these methods is implemented, then you should override the other as well

Persistent

What if you don’t override equals()?

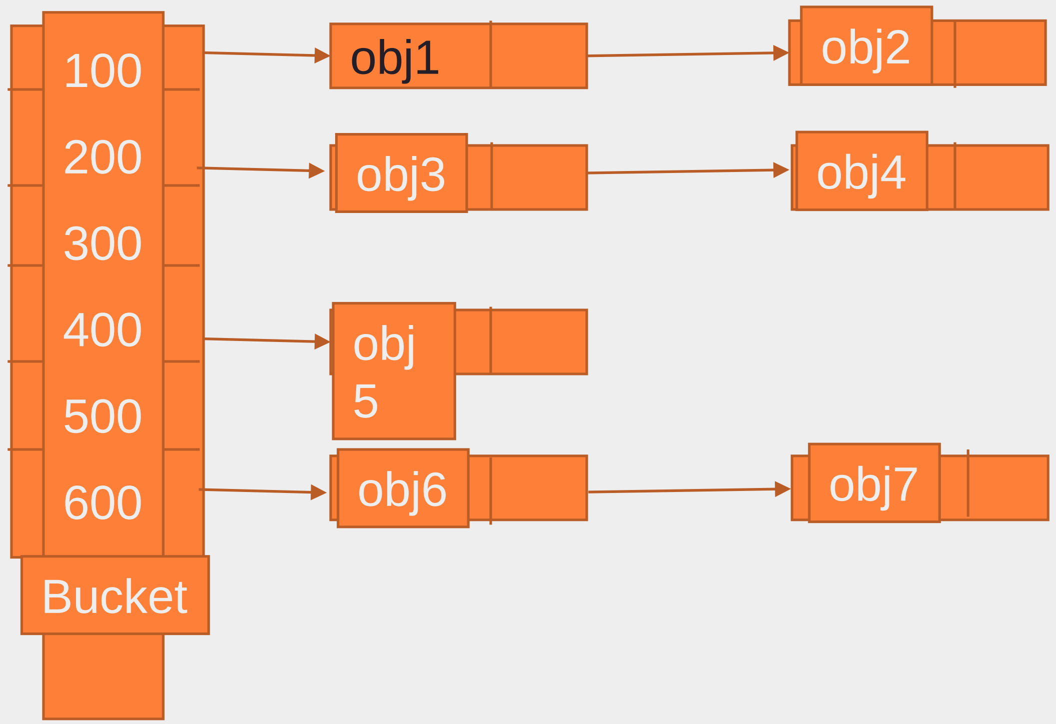
- Default behavior of equals() checks if 2 references are pointing to same object.
- Default behavior of hashCode() method generates unique code for every object.
- “Can not store objects in hashset or as a key in hashtable”.
- Default equals method in object class uses only the == operator for comparisons.
- Unless override equals, two objects are considered equal only if the two references refer to same object.

Deep dive into HashCode?

- Overriding only just the equals() method wont work with collections.
- So equals() and hashCode() methods are like contract.
- If we override equals we must override hashCode as well or viz. to make sure collections behave the same as they have defiened.
- Before storing the object in HashSet its HashCode is calculated and bucket is defiened to store that object.

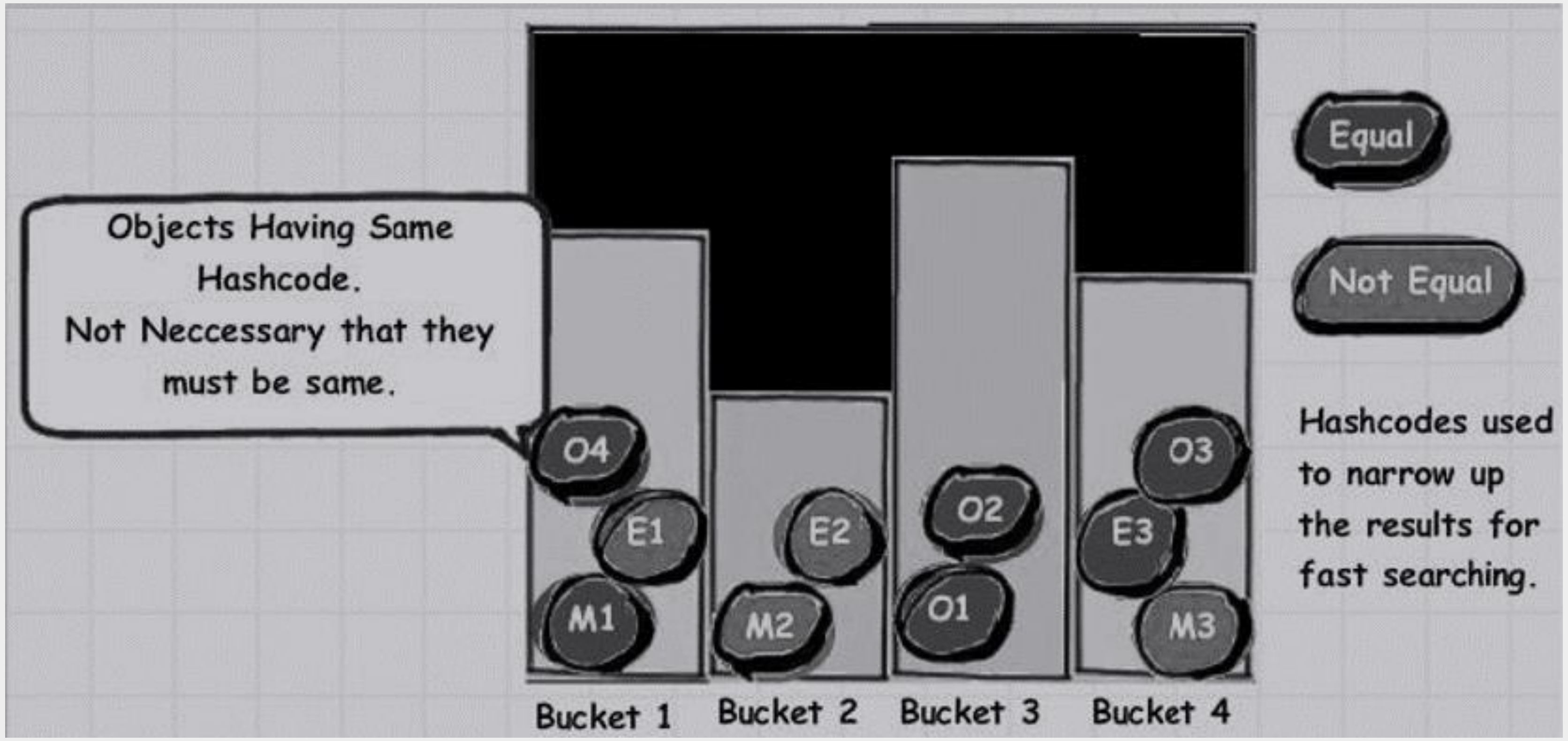
Deep dive into HashCode?

- In real life hashing, it’s not uncommon to have more than one entry in a bucket.
- Good hashing retrieval is typically a two-step process.
  - Find the right bucket depends on hashcode.
  - Search the bucket for the right element.



Persistent

Deep dive into HashCode?


Objects Having Same Hashcode.
Not Neccessary that they must be same.
O4
E1
M1
E2
M2
O2
O1
O3
E3
M3
Bucket 1
Bucket 2
Bucket 3
Bucket 4
Equal
Not Equal
Hashcodes used to narrow up the results for fast searching.

Storing into a HashSet…

- Two steps are involved while adding to HashSet:
  - HashCode is computed and the bucket of the object is identified.
  - Equals is executed to check if the object being added is equals to any contents in the same bucket.

Custom Sorting…!!!

- When we say sorting we consider two classes TreeSet or TreeMap.
- But they work for API classes i.e. String class and Wrapper classes
- So how do we sort in case of user defined objects like Employee, Vehicle, Student etc.
- To sort user defined objects we should use Comparable or Comparator interface.

Comparable…!!!

- Comparable is present in java.lang package
- Has only one method and one parameter :- int compareTo(T obj)
- Class itself must implement in order to compare objects
- Sorting is for only single datamember i.e at a time we can sort by employeeId or employeeName or employeeSalary.
- Preferred only when you want sorting by only single data member.

The collections framework comparisons with Comparable

```
public class Employee implements
Comparable<Employee> {
          ------------------------
          ------------------------
public int compareTo(Employee o) {

          if(getSalary() < o.getSalary())
          return -1;

          if(getSalary() > o.getSalary())
          return 1;

          else
          return 0;
          }}
```

The collections framework comparisons with Comparable

```
TreeSet<Employee> pList = new TreeSet<Employee>();

pList.add(new Employee("Ajay Kumar", 34673));
pList.add(new Employee("Vijay Kumar", 37847));
pList.add(new Employee("Sanjay Kumar",33335));
pList.add(new Employee("Dhananjay Kumar",74432));
pList.add(new Employee("Ranjay Kumar",24757));
```

Comparator…!!!

- Comparator is present in java.util package
- Has only 2 methods :- int compare(T obj1,T obj2) and boolean equals(Object obj)
- Any new class can implement comparator in order to compare objects
- Any number of datamembers can be sorted i.e at a time we can sort by employeeId, employeeName and employeeSalary.
- Preferred when you want sorting by number of data member.

Comparisons with Comparators

```
public class Employee {
        private String name;
        private int salary;

        public Employee(String n, int s) {
                name = n;
                salary = s;
        }

        public String getName() { return name; }
        public int getSalary() { return salary; }
        public String toString() {
        return getName() + " earns Rs." + getSalary();
        }
}
```

Comparisons with Comparators…….

```
public class SalaryComparator
          implements Comparator<Employee> {

          public int compare(Employee x, Employee y) {

          if(x.getSalary() < y.getSalary())
          return -1;

          if(x.getSalary() > y.getSalary())
          return 1;

          else
          return 0;

          }}
```

Comparisons with Comparators…….

```
TreeSet<Employee> pList;

pList  = new TreeSet<Employee>(new
SalaryComparator());

pList.add(new Employee("Ajay Kumar", 34673));
pList.add(new Employee("Vijay Kumar", 37847));
pList.add(new Employee("Sanjay Kumar",33335));
pList.add(new Employee("Dhananjay Kumar",74432));
pList.add(new Employee("Ranjay Kumar",24757));
```

Generics

- We already have gone through basics of generic, now we will be covering what are bounded types generics, and using them in method parameters, even we will cover wildcards in detail.

Generics bounded types

- Problem
  - Need a class which can be used as a container for instances of java.lang.Number or any of its sub-types. Class will have a method named *sum( )* which will invoke the *doubleValue( )* method on each java.lang.Number object.
- Solution
  - A generic class with a upper-bound type

Bounded types…..

T has been upper-bound to Number

```
public class NumberContainer<T extends Number> {

        private T values[];
        double sum= 0.0;

        public double sum() {
        for(T value : values)
        sum += value.doubleValue();
        return sum;
        }

        public double average() {
        return this.sum() / values.length; } }
```

Bounded types…..

```
NumberContainer<Integer> iObj = new
NumberContainer<Integer>();

NumberContainer<Double> dObj = new
NumberContainer<Double>();

double sum = iObj.sum();

NumberContainer<String> sObj = new
NumberContainer<String>();
```

❌

Wildcards

- Problem
  - The container already has two methods named *sum( )* and *average( )*. It needs a third method named *hasSameAverage( )* which can be used to compare the averages of two number containers.

- Solution
  - A method with wildcard argument

Wildcards…….

Method with wildcard argument indicating it can be passed any instance of NumberContainer<T>

```
public class NumberContainer<T extends Number> {
            --------------------------
            --------------------------
public boolean hasSameAverage(NumberContainer<?> r)
{
return this.average() == r.average() ? true : false;
}}

NumberContainer<Integer> iObj = new
NumberContainer<Integer>();

NumberContainer<Double> dObj = new
NumberContainer<Double>();
--------------------------
--------------------------
if(iObj.hasSameAverage(dObj))
            System.out.println("Their average is same.");
```

Reading, Writing, and Arithmetic

- How can we use Wildcards?
- What kinds of types does the compiler enforce for variables and arguments which referred to the type variables in the generic class?
- *Wildcard instantiations of generic types can be read as their upper bound and written as their lower bound.*
  - List<Date> can be read as the Date type but can be written as any subtype of Date

```
List< ? extends Date > someDateList = new
   ArrayList<MyDate>( );

   Date date = someDateList.get( 0 );//read as Object

List< ? > someList = new ArrayList<String>( );

 Object obj = someList.get( 0 );       //read as Object
```

Reading, Writing, and Arithmetic...

- Our wildcard instantiation has an upper bound of Date, we can read the type as Date
- Since we don’t know the correct type, the compiler won’t let us write anything to the List through our wildcard instantiation of the type

```
List< ? extends Date > someDateList = new
    ArrayList<MyDate>( );

    someDatelist.add( new Date( ) ); // compile time error

    someDatelist.add( new MyDate( ) ); //compile time error
```

Reading, Writing, and Arithmetic...

- We can write to the object as the lower bound type through our wildcard
- Not knowing what supertype of MyDate the elements are, we cannot read the list as any specific type

```
List< ? super MyDate > listAssignableMyDate = new
    ArrayList<Date>( );

//compile time error
listAssignableMyDate.add( new MyDate( ) );

listAssignableMyDate.add( new Date( ) );

Object obj = listAssignableMyDate.get( 0 );
                //read as Object
```

Generic methods

- A generic method which applies the linear search algorithm on arrays of any type.

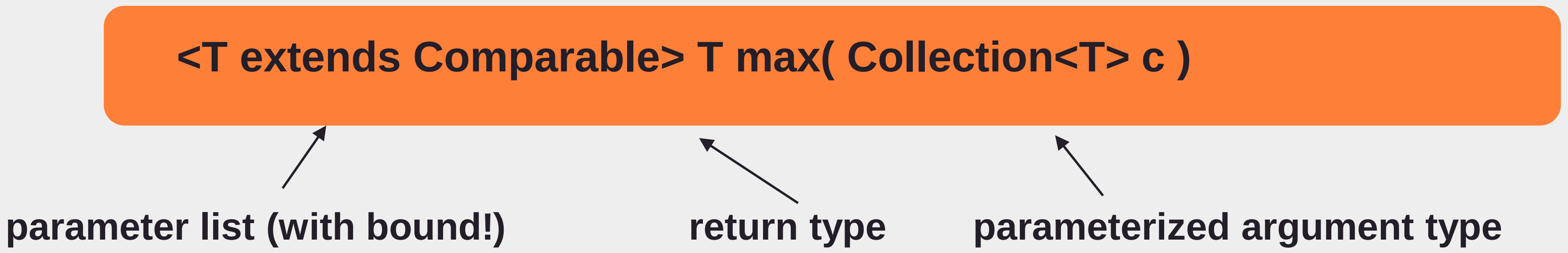
```
public class GenericMethods {
public static void main(String args[]) {
            String sArr[] = { "Good", "Great", "Fine", "Nice"
};

            Integer nArr[] = { 10, 20, 30, 40, 50 };
            if(LinearSearch("Jack", sArr))
            System.out.println("String found");

            if(LinearSearch(30, nArr))
            System.out.println("Integer found");

}
```

Generic methods

```
public static <T> boolean LinearSearch(T val, final T
arr[]) {

            for(T x : arr) if(x.equals(val))
            return true;

            return false;

}}
```

Generic Methods

- Classes and interfaces are not the only things that can be parameterized
- Methods can also be parameterized – we've seen one already: Collections.max()
- Here's a simplified version of its signature:



```
<T extends Comparable> T max( Collection<T> c )
```

Generic Methods

```
static void fromArrayToCollection(Object[] a,
    Collection<?> c)
  {
            for (Object o : a) {
            c.add(o);          //compile time error

            }
    }


static<T> void fromArrayToCollection(T[] a,
    Collection<T> c)
    {
            for (T o : a) {

            c.add(o);          //correct

            }

    }
```

Invoking Generic Methods

```
Object[] oa = new Object[100];

Collection<Object> co = new ArrayList<Object>();

fromArrayToCollection(oa, co);// T inferred to be Object

String[] sa = new String[100];

Collection<String> cs = new ArrayList<String>();

fromArrayToCollection(sa, cs); // T inferred to be

fromArrayToCollection(sa, co); // T inferred to be Object
```

Invoking Generic Methods

```
Integer[] ia = new Integer[100];

Float[] fa = new Float[100];

Number[] na = new Number[100];

Collection<Number> cn = new ArrayList<Number>();

fromArrayToCollection(ia, cn); // T inferred to be Number
fromArrayToCollection(fa, cn); // T inferred to be Number
fromArrayToCollection(na, cn); // T inferred to be Number
fromArrayToCollection(na, co); // T inferred to be Object
fromArrayToCollection(na, cs); // compile-time error
```

Interoperating with Legacy Code

- **Raw types :** ***The generic type without any type arguments.***
    - is assignment compatible will all instantiations of the generic type.
    - assignment of an instantiation of a generic type to the corresponding raw type is permitted ***without warnings***.
    - assignment of the raw type to an instantiation yield an "***unchecked warning***".

```
ArrayList    rawList    = new ArrayList();

ArrayList<String> stringList = new ArrayList<String>();

rawList    = stringList;         //unchecked warning

stringList = rawList;
```

One Instance of Static Field

Example (of a generic class with a static field):

Example (of several instantiations and usage of the static field(s)):

```
class SomeClass<T> {
        public static int count;
         ...
}
SomeClass<String> ref1 = new SomeClass<String>();

SomeClass<Long>   ref2 = new SomeClass<Long>();

    ref1.count++;                    // discouraged, but legal
    ref2.coun t++;                   // discouraged, but legal

    SomeClass .count++;        // fine, recommended
    SomeClass<String>.count++;                          // error
    SomeClass<Long>.count++;                            // error
```

A Generic Class is Shared by all its Invocations!

```
List<String> l1 = new ArrayList<String>();

List<Integer> l2 = new ArrayList<Integer>();

System.out.println(l1.getClass() = = l2.getClass( ));
```

Output ?

(true/false)

Types that can’t have Type Parameters

All types, except enum types, anonymous inner classes and exception classes, can be generic..

Automatic Type Inference in Generic object instantiation

- This feature is added in JDK1.7
- Usage of **Diamond Operator** (<>) .
- The diamond operator allows the way of declaring a new instance of a generic object without having to repeat its type parameters.
- E.g. List<String> list = new ArrayList<String>(); //without using diamond operator
  List<String> list = new ArrayList<>();//with diamond operator

FAQ

- Describe collection framework in detail?
- Using Queue and Deque interface.
- How Navigable set and Navigable Map works?
- Why to override equals and HashCode?
- What is Hashing?
- When to use comparator and comparable?
- Using Bounded Types generics and Wildcards?

Summary

With this we have come to an end of our session, where we discussed about

- Some new classes and interfaces of Collection framework
- Bounded type and wildcard generics

At the end of this session, we see that you are now able to answer following questions:

- Using Queue and Deque?
- What is hasing and how it works for Hashset?
- Bounded type and wildcard in generics?

Appendix

References

Thank you

Reference Material : Websites & Blogs

- https://docs.oracle.com/javase/tutorial/collections/intro/
- http://beginnersbook.com/java-collections-tutorials/
- https://docs.oracle.com/javase/tutorial/collections/interfaces/deque.html
- http://www.tutorialspoint.com/java/util/java_util_arraydeque.htm
- http://www.mkyong.com/java/how-to-work-with-java-6s-navigableset-and-navigablemap/
- https://www.ibm.com/developerworks/library/j-jtp05273/
- https://docs.oracle.com/javase/tutorial/java/generics/boundedTypeParams.html
- https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html

Reference Material : Books

- **Head First Java**
  - By: Kathy Sierra, Bert Bates
  - Publisher: O’Reilly Media, Inc.
- **Java Complete Reference**
  - By Herbert Schildt

Persistent

Thank you!

Persistent Interactive | Persistent University